Informatics

Automated soundness testing of program analyzers

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Markus Fleischmann, BSc.
Matrikelnummer 11813846

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in sc. Maria Christakis
Mitwirkung: Dr. sc. Valentin Wüstholz
Univ.Ass. Dipl.-Ing. David Kaindlstorfer, BSc.

Wien, 20. Jänner 2024

Markus Fleischmann	Maria Christakis

Informatics

Automated soundness testing of program analyzers

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Markus Fleischmann, BSc.
Registration Number 11813846

to the Faculty of Informatics

at the TU Wien

Advisor:	Univ.Prof.in Dr.in sc. Maria Christakis
Assistance:	Dr. sc. Valentin Wüstholz
	Univ.Ass. Dipl.-Ing. David Kaindlstorfer, BSc.

Vienna, 20th January, 2024

Markus Fleischmann   Maria Christakis

Erklärung zur Verfassung der Arbeit

Markus Fleischmann, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Jänner 2024

Markus Fleischmann

Danksagung

Als Erstes möchte ich meiner Betreuerin, Maria Christakis, für ihre stetige Führung und feste Unterstützung beim Schreiben dieser Arbeit bedanken.. Ohne ihre undendliche Motivation und ihren immerwährenden Optimismus wäre diese Arbeit vermutlich nie fertig geworden. Einen großen Dank auch an meine mitwirkenden Betreuer, Valentin Wüstholz und David Kaindlstorfer für ihre vielen Bemerkungen und Ideen, ohne die es diese Arbeit in ihrer derzeitigen Form auch nicht gäbe.

Zuletzt möchte ich auch meiner Familie und meinen Freunden danken, die mich im vergangenem Jahr stets unterstützt haben, selbst wenn mich meine Arbeit oft daran gehindert hat, ihnen die Aufmerksamkeit zu widmen, die sie verdienen.

Acknowledgements

I would like to thank my supervisor, Maria Christakis, for her excellent guidance and firm support throughout the writing of this thesis. It was thanks to her constant optimism and encouragement that I was able to complete this work. I would also like to thank my co-supervisors, Valentin Wüstholz and David Kaindlstorfer, for providing many great ideas and insights, without which this work would not exist.

Lastly, I would also like to thank my family and friends for supporting me throughout the last year, even when my work kept me from spending as much time with them as I would have liked and they would have deserved.

Kurzfassung

Programmanalysestools werden häufig eingesetzt, um sicherzustellen, dass Programme fehlerfrei sind. Fälle, in denen diese Tools mögliche Fehler übersehen (Fehlschlüsse), sind daher besonders kritisch. Frühere Methoden, Programmanalysetools zu testen, verließen sich auf entweder auf differentielles Testen, oder testeten hauptsächlich die Genauigkeit dieser Tools. Wir präsentieren einen Ansatz, der es ermöglicht fehlerhafte Programme mit Hilfe von erfüllbaren SMT Formeln zu generieren. Diese Programme eignen sich wiederum um automatisiert zu testen, ob Programanalysetools mögliche Fehler erkennen, ohne dass dadurch differentielles testen erforderlich wird. Um zu prüfen, ob unser Ansatz Fehlschlüsse findet, implementierten wir ihn in einem Tool namens Minotaur und verwendeten dieses, um Fehler in 8 moderne Programanalysetools zu suchen. Über die letzten 9 Monaten haben wir Fehlschlüsse in 5 dieser Tools gefunden. Wir erklären und kategorisieren diese, um zu verstehen welche Art von Fehlern unsere Technik finden kann.

Abstract

Program analyzers are commonly used to ensure that programs do not contain errors. Cases where they miss an error (soundness bugs), are therefore especially critical. Past techniques for testing program analyzers have either relied on differential testing, or focused mainly on precision issues. We present a technique that generates unsafe programs using satisfiable SMT formulas. These can be used to test soundness of program analyzers in an automated fashion, without the need for differential testing. In order to test the technique, we implemented it in a tool called Minotaur, and used it to look for soundness bugs in 8 state-of-the-art program analyzers. We found bugs in 5 of the tools, which we explain and categorize in order to analyze the types of bugs our technique can find.

Contents

CHAPTER 1

Introduction

Over the last decades, tools that perform program analysis have become more commonplace in software development, especially in safety-critical areas. They can prove a broad range of program properties, such as the reachability of errors and the safety of memory accesses. Therefore, they play an important role in ensuring that code remains bug-free and secure.

Nevertheless, program analyzers are themselves programs and can also contain bugs. These can cause them to crash, get stuck in an endless loop, or, more interestingly, return an incorrect result. We distinguish two types of wrong results: We call the cases where the program is *safe* (correct), but the analyzer reports an error *precision issues*, and refer to the case where the program is *unsafe* (contains an error), but the analyzer claims correctness *soundness bugs*.[1]

In our opinion, soundness bugs are especially dangerous, as they are hard to notice in practice, and can lead to a false sense of security towards the analyzed programs. On the flip side, this means that fixing even a few soundness bugs in program analyzers can prevent errors in a large number of programs.

1.1 Previous work

Testing program analyzers can have an outsized impact, and a few approaches for doing this in an automated fashion have already been developed. Any automated testing has to find a solution to the *oracle problem* [2], which involves finding the expected results for automatically generated test cases, and previous approaches have relied on well-known techniques for resolving this:

[1]To further avoid confusion, we use *test*/*program* and *error* when discussing the analyzed code, and reserve *tool* and *crash*/*bug*/*issue* for talking about the program analyzers themselves.

MCFuzz [72] uses a different approach, known as *metamorphic testing* [75]. It takes safe seed programs and inserts errors in such a way that the safety of the resulting program remains known. Alas, the majority of generated programs are safe, meaning that most of the test-cases can only check analyzer precision.

1.2 Contributions

Our approach focuses on generating unsafe programs that can be used to test the soundness of analyzers, as we deem soundness to be more important than precision. In particular, we make the following contributions:

- We present a technique to generate unsafe C programs using satisfiable SMT formulas as well as a minimization routine, which generates minimal explanations for cases where an analyzer fails to find an error. As the programs are known to be unsafe, the oracle problem becomes trivial when using them as test-cases.
- We implement the generation and minimization technique in a tool that automatically generates programs and uses them to test program analyzers.
- We performed extensive testing on 8 state-of-the-art program analyzers and managed to find a total of 10 soundness bugs in 5 tools. Most confirmed bugs were quickly fixed. We also explain and categorize the bugs to analyze which types of bugs our approach can find.

1.3 Outline

Chapter 2 gives some more details on program analyzers and presents the theory and tools on which we build our generation approach. Chapter 3 then goes into detail on how we generate programs and ensure that they are sufficiently varied. It also briefly outlines a way to generate minimal versions of programs that trigger bugs, which helps developers spot the cause quickly.

In Chapter 4 we discuss the tools we tested and how we performed the tests. Chapter 5 gives a brief overview over the test results, before concretely discussing what caused the bugs we found.

Finally Chapter 6 presents some related work, while Chapter 7 gives a brief conclusion and looks at different ways in which our approach could be improved.

CHAPTER 2
Preliminaries

In this chapter we recap some of the basic notions required for this thesis.

2.1 Program analyzers

Program analyzers are tools that can automatically analyze properties of programs. Most often these properties have to do with some notion of safety, such as:

- Reach-safety: Check that a given (error-)function is not reached
- Assertion-safety: Check that an assertion does not fail
- Termination: Check that program execution terminates
- Undefined behaviour: Check that no undefined behaviour occurs. This is language specific and, in the case of C, includes e.g. invalid memory accesses, overshifts, (signed) oveflows and division by zero.

We call a program *safe* w.r.t. a property if the property holds for all executions, and *unsafe* if there is an execution that violates the property.

Many of these properties can be reduced to one another: For example, division-by-zero safety for the statement `l/r;` can be checked as assertion-safety of

```
assert(r!=0);
```

which in turn can be expressed as reachability:

```
if(r==0) error();
```

An important notion for analyzers is *nodeterminism*. This is used to model values that might differ for every execution of the program, such as user input or sensor data. Typically analyzers provide a way to declare variables as non-deterministic. This means that the analyzer can freely pick the value of the variable when trying to find an error and, equivalently, has to consider all possible values when trying to prove safety.

2.1.1 Types of Analyzers

We first define the notions of *soundness* and *precision* mentioned in Chapter 1:

A program analyzer is *sound* if it never claims safety for unsafe programs.
A program analyzer is *precise* if it never claims errors for safe programs.

Note that analyzers are allowed to return *unknown*, if the technique cannot determine the safety of a program. This is sometimes preferred, as it maintains precision and soundness.

There are various approaches to program analysis, with trade-offs in runtime, precision and soundness: *abstract interpretation (AI)* [19] sacrifices precision in order to improve scalability, while maintaining soundness. Model checking [15, 18], on the other hand, tries to be precise at the cost of runtime and sometimes (e.g. in bounded model checking (BMC) [9]) only guarantees partial soundness.

Often times a distinction is also made between *static* [40] and *dynamic* analysis, where dynamic analysis performs analysis on a running program, while static analysis only reasons about program code.

The tools we tested mainly perform various forms of static analysis. However many of the tools we tested (see Section 4.1) implement and even combine several different approaches, including dynamic analysis techniques. This makes an exact classification difficult.

2.2 SMT reasoning

Satisfiability modulo theories (SMT) extends the satisfiability problem over propositional formulas (SAT) with theory reasoning. The theories can take many shapes, as long as they have a decision procedure and some operator with boolean result. We call theory expressions that yield a boolean result *theory atoms*. SMT *formulas* are built from theory and boolean atoms using standard propositional connectives. This allows for more expressive formulas than SAT, while still allowing for relatively efficient reasoning [52, 62], depending on the theories being used.

Recall that in SAT solving a formula is *sat* if it has a model (an interpretation that makes the formula true) and *unsat* otherwise. For SMT we distinguish propositional and theory satisfiability: for example the formula

$$(y \leq x \rightarrow x < y) \land (x = 2 \lor \neg(y = 3)) \land y = 3$$

is propositionally sat: clearly $y = 3$ must be true (as it is a unit clause) and by unit propagation $x = 2$ must also be true. Lastly, to make the implication true, we can set $y \leq x$ to false. Setting $y \mapsto 3, x \mapsto 2$ also yields a valid model in the theory of integer arithmetic. Therefore this formula is sat both propositionally and for the theory of integer arithmetic.

Meanwhile the formula

$$(y \leq x \rightarrow x < y) \wedge (x = \mathbf{4} \vee \neg(y = 3)) \wedge y = 3$$

has a similar propositional model, but is unsat in the theory of integer arithmetic: setting $y \leq x$ to false gives $3 \not\leq 4$, which is a contradiction. Setting $y \leq x$ to true means $x < y$ must also hold, but $4 < 3$ is also a contradiction.

In the following we will not distinguish between propositional and theory satisfiability, but rather call an SMT formula *sat* if it is both propositionally and theory satisfiable, and *unsat* otherwise.

Over the past years many SMT solvers have been developed, which combine propositional and theory reasoning to automatically decide SMT formulas. Popular solvers include Z3 [48], mathSAT [14], Yices [23] and CVC4 [5], though there are many more [53, 13, 59]. In addition to solving the decision problem (whether a given formula is sat or unsat), modern solvers provide a broad range of further features, such as unsat core extraction (finding a minimal explanation for an unsat formula) or formula simplification.

2.2.1 SMT-LIB

SMT-LIB [4] is an effort to provide a standard notation for SMT formulas and problems. The standard is implemented by most modern SMT solvers and enables competitions such as SMT-COMP [63] and APIs like pySMT [28], both of which have been useful for this thesis.

SMT-LIB defines a wide family of logics, named after the theories they include. In this work we mainly use the theory of quantifier-free (non)linear integer arithmetic (`QF_LIA` / `QF_NIA`), as well as the theory of quantifier-free fixed-size bitvectors (`QF_BV`). The latter defines reasoning over bitvectors of finite length and is particularly useful for modeling machine arithmetic. We also use the extension of those theories with the theories of arrays (`A`) and uninterpreted functions (`UF`). For a full definition of these logics, please refer to the SMT-LIB website [3].

2.2.2 STORM

STORM [45] is a tool that automatically tests SMT solvers for critical bugs (cases where the formula is *sat* but the solver returns *unsat*). It does this by generating SMT formulas which are satisfiable by construction:

Given an SMT formula as seed, STORM uses an SMT solver to obtain a model $\mathcal{M}$ for it. It then computes the truth-value of every subformula under $\mathcal{M}$. These subformulas are combined to create new formulas (*mutants*), while incrementally computing their truth values (still under $\mathcal{M}$). It can then output a set of true mutants, which must clearly be satisfiable (as $\mathcal{M}$ is a model and they are true under $\mathcal{M}$).

STORM also provides the infrastructure to automatically test SMT solvers using the generated mustants, as well as a minimizer. In this work we use STORM to generate a pool of satisfiable formulas from a (potentially unsatisfiable) seed.

2.3 Fuzzle

Fuzzle [33] is a tool for benchmarking fuzzers. It generates programs from mazes (see Section 3.1 for more details), through which tested fuzzers need to navigate in order to find an error. This allows for nice visualisations both for the problem itself and the coverage the fuzzers managed to achieve.

In order to improve realism the authors selected a few bugs from the national vulnerability database [51] and translated the error traces first to SMT formulas (using Klee [12]) and finally to if-guards. These ensure that the fuzzers have to find the values that match the error trace of a "real" bug, in addition to finding their way through the maze.

Fuzzle also implements automatic benchmarking of state-of-the-art fuzzers, including benchmark generation, testing, as well as computation and visualization of coverage statistics. In our approach Fuzzle provides the basic framework on which we built our program generation.

CHAPTER 3

Generating unsafe programs

We now present our approach for generating programs: First, an adapted version of Fuzzle is used to generate scaffolding code, which contains an error. This scaffold is then populated with logic obtained from an SMT formula, ensuring that the error is reachable iff the formula is sat.

To increase diversity of the programs we propose different transformations of the scaffold, which maintain reachability of the error. STORM is run on SMT seeds to generate satisfiable formulas, both to boost variety and ensure that the errors are always reachable.

Lastly a simple minimization procedure can be performed on cases which caused a bug. It generates a minimal program, which is guaranteed to still trigger the bug, while eliminating non-relevant code.

3.1 Program generation

Program generation is based on Fuzzle, which we adapted to follow SV-COMP norms [7] for error calls and declaration of nondeterministic variables. Fuzzle first generates a scaffolding, consisting of branching function calls and injects logic via the branching conditions. In our case the guards for these branches will be generated using SMT files as seeds.

3.1.1 Scaffolding

The scaffolding of our programs can be seen as a 2D-maze on a square grid. These mazes are generated randomly, with each maze having a *start*, an *exit* and exactly one path connecting them, which we shall call the *solution*. The start represents the program entry point (usually the `main()` function), while the exit marks the error function call (`__VERIFIER_error()` in SV-COMP).

Mazes are first translated into directed graphs, with an edge existing between two nodes iff they are neighbouring in the maze (i.e. are next to each other in the grid and don't have a wall between them). To generate a scaffolding these nodes are then rendered as functions, such that a function f_1 can call a different function f_2 if there is an edge from f_1 to f_2 in the graph. In case there are several outgoing edges, if-branches are added to ensure that only one function is called:

```
function_1(){
    if(___){
        function_2();
    } else if(___) {
        function_3();
    } else if(___) {
        function_4();
    } else {
        //
    }
    return;
}
```

Listing 3.1: Example scaffolding for a function with 3 neighbours.

By design of these translations there must be a sequence of function calls, corresponding to the solution of the maze, that starts at the main function and finally calls the error function. This ensures that the error is indeed reachable.

3.1.2 Transformations

To further increase the variance of programs, we also propose some transformations on the scaffolding. All transformations are designed to preserve reachability of the error call.

Remove backwards edges: (Partially) remove backward edges. Backwards is defined via a breadth-first search from the maze start, where an edge from node n to node m is backwards iff n is found first by the search.

In particular, removing all backwards edges results in a non-recursive program, as there is no way to revisit a node that has been visited before. This is required for some analyzers that do not support recursion.

This transformation preserves reachability as the solution only uses forward edges. Thus, removing backwards edges cannot make the solution invalid.

Removing walls: Removing certain walls in the maze, makes nodes have more neighbours and allows for more paths (and potentially easier) paths through the maze.

This preserves reachability as removing walls cannot make the original solution invalid.

Chaining mazes: Make the exit of one maze lead to the start of the next maze. This allows chaining both an arbitrary number of mazes and mazes of arbitrary shapes.

This preserves reachability by transitivity of reachability: Since the exit of any maze m_n is reachable from its start, and the start of maze m_{n+1} is reachable from maze m_n then the exit of maze m_{n+1} must be reachable from the start of maze m_n. And clearly the start of maze m_1 is already reachable, as it is the program entry point.

3.1.3 Guard generation

Similar to Fuzzle, there are two main ways we generate expressions to populate if-guards:

Default generation: By default the guards partition the value range of a character ($[-128, 127]$) into sections of equal size, depending on the numbers of neighbouring functions (eg. "$c < -43$", "$c < 42$", "$c \geq 42$" for three neighbours). This character is declared as a non-deterministic `char` (by calling the function `__VERIFIER_nondet_char()` from the SV-COMP interface). This means that program analyzers can freely choose the value of c and, by extension, the next cell to visit. This allows tools to navigate their way through the maze.

SMT based generation: Special attention is given to those branches that are part of the solution. To reach the corresponding branch, the analyzer first needs to satisfy a series of if-statements. Should all the if-statements be satisfied a flag is set that allows for calling the next function.

The if-guards are populated from the given SMT file, by translating the clauses of the formula into boolean C-expressions (see Section 3.1.4). As the SMT seed is satisfiable every clause must be satisfiable as well. Free variables in the formula are declared as non-deterministic variables of the given type, similarly to c in the default case.

The clauses of the seed formula are spread evenly across the different functions of the solution. But, due to scope limitations, all clauses with shared variables must be contained within the same function. This means some functions might still be significantly larger than others.

A function call, populated with the SMT clauses $\{x < 3, x > 2 \vee x - 2 = 0\}$ using the theory of integer arithmetic (`QF_LIA`), could look as follows:

```
function_1(){
    // Pick variables
    char x = __VERIFIER_nondet_char();
    long c = __VERIFIER_nondet_long();

    // Translated SMT logic
    int flag = 0;
    if (x < 3){
        if ((x > 2) || ((x-2)==0)){
            flag = 1;
        }
    }

    if(c < -43){
        function_2();
    } else if(flag == 1 && c < 42) {
        // Next step to the solution
        function_3();
    } else if(c >= 42) {
        function_4();
    } else {
        //
    }
    return;
}
```

Listing 3.2: Populated function

3.1.4 Translating SMT-LIB to C

We propose an extension of the original Fuzzle SMT parser to cover most quantifier free formulas over integers or bitvectors, extended with arrays and uninterpreted functions (QF_AUF[LIA/NIA/BV]). Some expressions are handled in an incomplete manner, but all translations are designed to maintain equisatisfiability with the original formula.

If a clause cannot be translated, a warning is raised and `1` (i.e. true) is returned. This at least guarantees that the error remains reachable.

Integers

SMT-LIB integers are unbounded, while C integers are bounded within the range $(-(2^{63}), 2^{63}]$ (assuming `long` integers on 64 bit systems). It could be the case that a seed is only satisfiable using values outside this range (e.g. the clause $x > 2^{63}$), which would make the translated C-expression unsatisfiable.

To ensure that the error is always reachable, we generate constraints $\{f >= -(2^{63}), f < 2^{63}\}$ for every subformula f of type integer in the seed, and check if the seed remains satisfiable given those constraints.

As we are using signed integers, program traces that overflow might have undefined behaviours. However, if the range checks succeed, there must be at least one error trace without overflows. This means that the generated programs can also be used to test analyzers (like some abstract interpreters) which ignore overflows by design.

Bitvectors

If we want to generate well-defined programs, we can use SMT-LIB bitvector formulas. Bitvectors are modelled using C integer types. By default unsigned integers are used, as they correspond more closely to SMT bitvectors. In particular, unsigned integer overflow in C is well defined, whereas signed overflow is not. A bitvector of width m is always assigned to the smallest C type of size $\geq m$. While most of the translations are quite straightforward, there are some details that should be mentioned:

- **Casting:** When casting to sizes m that have a matching type in C (8,16,32,64) we simply cast to that type. Otherwise, for unsigned casts we compute the bitwise-and with $2^m - 1$. For casts to signed, we need to check if the m-th bit of the bitvector is `1` (meaning the result should be negative) and, if so, explicitly subtract 2^m from the unsigned value before casting.

- **Automatic upcasting:** C performs automatic upcasting of types, where the result of an arithmetic operation can be of a different type than the operands, if the result does not fit in their type. SMT-LIB bitvectors usually stay the same size, and simply overflow modulo 2^m. Therefore we have to recast the result of arithmetic operations and left-shifts, to ensure that the size of the resulting bitvectors also remain the same in the C program.

- **Shifting operations:** According to the C standard, shifts of length greater than the current type (e.g. shifts > 32 for ints) are undefined. Therefore we only allow shifts by a constant amount, which we can check to be in range for the current type.

- **Division by zero:** Division by zero is well-defined in SMT-LIB QF_BV so we use helper functions for signed/unsigned division/remainder operations to handle the case where the right operand is zero.[1]

- **Allowing undefined behaviour:** Optionally, we can also accept undefined behaviours in our programs. In that case we can simply translate division and remainder as `/`, `%` and perform variable-sized shifts. However to ensure that there still is an error trace without undefined behaviour, we add the corresponding constraints and check satisfiability beforehand, similar to the integer case described earlier in this section.

Functions

At the moment our approach only supports function calls with constant parameters, by modelling a function call `f(c1,c2,...,cn)`, with constants `c1,c2,...,cn`, as a non-deterministic variable `f_c1_c2_..._cn`. The type of this variable corresponds to the codomain of f.

Arrays

The semantics of arrays in C and SMT-LIB differ in a few points. This means that extra care is needed when performing translations:

[1]While it would be possible to perform this inline via the ternary `?:` operator, it would require rendering the right operand twice (for the zero check and the operation) and could lead to an exponential blowup of the expression. For the same reason we also employ helper functions for signed casts and bitvector `rotate` expressions.

- **Array size:** While SMT-LIB arrays have arbitrary size[2], C arrays are fixed size and, ideally, not larger than necessary. Therefore we compute a minimum required array size for the seed: Constraints are added ensuring that all indices are less than a given maximum i_{max} (and > 0 for integer logic). Should the formula become unsatisfiable because of this, we iteratively double i_{max} until the formula becomes satisfiable. If i_{max} become too large, we give up and return an error.

  If we want to generate a well-defined program, these constraints are also translated to C, which ensures that no invalid memory accesses can occur.

- **Array declaration:** Reading an uninitialized value of an array in SMT-LIB gives a non-deterministic value, but is undefined in C. Therefore, when declaring an array, we introduce a loop that explicitly sets every index of the array to a non-deterministic value.

- **Array comparison:** Array equality is defined as element-wise equality in SMT-LIB. As our arrays are of equal and fixed size, this can easily be translated to a fixed-size loop.

- **Array stores:** SMT-LIB array stores are local operations, in the sense that they only affect reads and compares which are ancestors in the formula tree. Simply translating `(array-store a i v)` as `a[i]=v` can cause problems, as the latter can also affect reads in sibling branches, if they are rendered later in the program.

  To solve this we transform the original formula as follows: a new array a' is introduced for every store (or sequence of stores) to an array a. Then a is replaced by a' in any ancestors of the store. Furthermore we add a constraint ensuring that $a = a'$, thereby maintaining equisatisfiability of the formulas.

  Once the formula has been translated, sibling stores are guaranteed to be to different arrays. Therefore stores can now safely be encoded as `a'[i]=v` using a helper function. The equality constraints can be translated as mentioned above, but have to be rendered before other clauses, to ensure that the comparison happens before the new array is modified.

- **Multi-dimensional arrays:** As C stores multi-dimensional arrays in a continuous block of memory, we can view an m-dimensional array of size n as an array of size n^m. Using our knowledge of the original dimensionality, we can calculate the corresponding indices for the flattened array.

  Reading a multi-dimensional array in SMT-LIB array also yields an array (of lower dimension), which can be simulated by simply moving our pointer by $i * nm - 1$. We also split stores into value stores (as described above) and array stores, where an entire array is stored at once. Initialisation and comparison stay the same, except they run from 0 to $n^m - 1$.

[2]Actually SMT-LIB array indices are typed and can therefore be bound by the size of this type. However, even if this type is finite, it would usually be too large for analyzers to handle.

3.2 Generating SMT formulas

We use STORM (see Section 2.2.2) to increase the number of available SMT formulas and to ensure that all formulas, from which we generate guards, are satisfiable. In order to select the seeds on which we run STORM, we first check whether they are understood by our parser. As STORM does not generate any new atoms, in most cases any generated mutants are also supported.

3.3 Minimizer

Programs generated by our approach can be very large. Therefore we present a minimizer, which generates minimal explanations for soundness bugs. The approach is quite simple an relies on the assumption that most bugs are caused by just a hand full of clauses. However our experience so far has shown that this assumption seems to hold in practice (see Chapter 5.2).

Minimization happens on two layers. First, we try to eliminate the scaffolding, by rendering the error and all translated SMT clauses in a single function call. If the analyzer still does not find the error this is maintained for the second minimization step, otherwise no minimization of the scaffolding is performed.

As a second step we minimize the number of clauses used for generating guards (see Algorithm 3.1). We repeatedly drop either half of the clauses, until the error is found by the analyzer in both resulting programs. At that point we check if any individual clause can be dropped while maintaining the bug.

After checking all clauses this way, we arrive at a minimal program, in the sense that all remaining clauses are relevant to the bug. While this does not always find the smallest possible working example, in our experience the results have proven small enough to quickly spot the causes of bugs.

Algorithm 3.1: Minimization of clauses

```
$C \leftarrow formula$; // Initialize clauses from formula
$bug \leftarrow \top$; // Analyzer missed bug
$firstHalf \leftarrow \top$; // Which half to keep

// Try to remove half of the clauses at a time
while $bug \lor \neg firstHalf$ do
    $C' \leftarrow drop\_half(firstHalf)$;
    $bug \leftarrow analyzer\_misses\_error(C')$;
    if $bug$ then
        $C \leftarrow C'$;
        $firstHalf \leftarrow \top$;
    else
        $firstHalf \leftarrow \neg firstHalf$;
    end
end

//Switch to individual clauses
for $c \in C$ do
    if $analyzer\_misses\_error(C \setminus \{c\})$ then
        $C \leftarrow C \setminus \{c\}$;
    end
end
return $C$;
```

CHAPTER 4

Experimental setup

We implemented our approach in a python tool called Minotaur [1] forked from Fuzzle. In addition to generating and minimizing programs, Minotaur allows for configurable and automated testing of program analyzers.

4.1 Tested tools

We selected 8 tools/frameworks to test using Minotaur:

- Ultimate: a framework offering various program analysis toolchains. We tested Kojak [24](a software model checker (SMC)) based on CEGAR [16]), Taipan [30](AI [19] over program paths), Automizer [34] (an SMC based on trace abstraction and automatas) and GemCutter [25](an SMC that can also verify concurrent programs).
- CPA-Checker [8]: a configurable software verifier that provides various types of analysis. We tested its value- and predicate-analysis [1, 29], k-induction [21], symbolic execution [37], BMC [9] and the symbolic memory graph analysis [22].
- SeaHorn [39, 32]: a framework which provides combinations of BMC, model checking via constrained Horn clauses (CHCs) and AI. We mainly tested the BMC engine, as the AI approach does not support bit-precise reasoning well.
- ESBMC [27]: a bounded-model checker based on SMT solving. Apart from the default BMC engine it also supports incremental BMC, k-induction and interval-analysis for invariant generation.

[1] At the time of writing the source code is in a private repository. It will be made available in the future at `https://github.com/Fleischmaki/Minotaur`.

- MOPSA [35, 47]: an abstract interpreter focused on scalability over precision. It provides a wide range of domains and a custom contract language for specification.
- Symbiotic [58]: Symbiotic models program properties via finite-state machines, but uses slicing and symbolic execution to eliminate potential imprecision.
- CBMC [17]: a bounded-model checker. It unrolls programs and translates them into SMT formulas over bitvectors.
- 2ls [55]: 2ls extends BMC with k-induction and invariant generation to (dis-)prove various safety properties, including termination. We test both the individual components, as well as the combined k-induction k-invariant analysis.

All of the analyzers implement the SV-COMP interface and participated in the latest edition of SV-COMP [6] (with the exception of SeaHorn, which last participated in 2016). All tools were rebuilt regularly to ensure that any bugs found would be on the latest version of the tool.

4.2 Parameter and seed selection

Machine-architecture was set to 64 bits for all tools. As the only loops contained in our programs are over arrays, unroll bounds, if present, were set large enough to handle the array size of the generated programs (see Section 3.1.4). Function inlining was enabled if it proved necessary to handle our helper functions.

Seed files where taken from the SMT-COMP benchmark sets [26] for the supported logics and prechecked, as discussed in Section 3.2. We further discarded seeds with very few atoms, as this usually means that those atoms are very large, which affects the usefulness of both STORM and the minimizer. Lastly we required that satisfiability is found within 30 seconds, as our approach can require several satisfiability checks, e.g. when computing the minimum array size. This resulted in a total of roughly 20.000 seeds for our experiments.

We settled on 15 transformations (i.e. different SMT mutants and scaffold-transformations; see Sections 3.2 and 3.1.2) per seed file, as this proved to be enough to have most bugs occur several times per seed. STORM's parameters were fixed to generate small mutants, as otherwise tools would take too long to run. When the seed files used bitvector logic, we only tested configurations which support bit-precise reasoning.

In accordance with the approach of swarm testing [31], we tried to randomize any remaining parameters (both for the tools tested and for Minotaur) as much as possible.

4.3 Hardware

The tests were run on the ikarus server running Debian 6.1.55-1 with an AMD EPYC 7702 64-Core processor and 512GB of RAM. All tools were containerized to allow for parallel execution. The time limit per test-case was set to 60 seconds and each container received 4GB of memory.

CHAPTER 5

Experimental results

5.1 Testing

We started testing with early versions of the tool in mid-March 2023 and ran tests until the end of December 2023, giving a total time span of roughly 9 months. Most of the time was spent testing with bitvector logic, as support for integer logics was only added later in development.

5.2 Found bugs

We were able to find a total of 13 issues in 5 different tools (see Table 5.1). Out of these 10 were soundness bugs, 2 were precision issues and one caused crashes. With the exception of bugs found in Symbiotic (which no longer seems to be actively supported), all bugs were confirmed and all soundness bugs (except the most recent one) fixed.

The minimizer has been a good tool for generating small explanations, with one developer commenting

> "Thanks for reporting this! [...] the minimal program here makes it easier to investigate."[64]

and answering to another issue:

> "Thanks for this greatly written bug report with a perfect MWE [minimal working example]! Using this I was quickly able to identify the relevant bug". [65]

Tool	Component	Issue Nr.	Bugtype	Cause	Staus
CPA	k-induction	1114	Soundness	Overflow	Fixed
Ultimate	Framework	642	Soundness	Overflow	Fixed
CPA	k-induction	1130	Soundness	Operators	Fixed
Symbiotic	-	246	Soundness	Arrays*	Open
Symbiotic	-	247	Soundness	Arrays*	Open
Ultimate	Automizer/GemCutter	646	Soundness	Operator Translation	Fixed
Ultimate	Kojak	647	Crash	-	Fixed
ESBMC	Interval-Analysis	1363	Soundness	Overflow/Casts	Fixed
ESBMC	Interval-Analysis	1392	Soundness	Casts	Fixed
MOPSA	-	150	Precision	Casts	Confirmed
CPA	k-induction	1194	Soundness	Operator Translation	Fixed
MOPSA	-	157	Precision	Domain/by Design	Confirmed
ESBMC	Interval-Analysis	1565	Soundness	Typecasting*	(partially) fixed

Table 5.1: List of bugs found by Minotaur. Causes with * are best guesses. Precise causes, if known, are explained in Section 5.3f.

Indeed none of the bugs were caused by the mazes structure and most were caused by one or two clauses. This meant that the minimizer could drop the majority of clauses and produce very small explanations.

5.3 Bug causes

The programs quoted here were further cleaned up manually before reporting, but the logic causing the bug is the same as in the original (minimized) version. We try to categorize the bugs into a few distinct categories, depending on their cause:

5.3.1 Incorrect operator semantics

Some of the bugs where caused because the analyses did not soundly model the semantics of C operators. In Ultimate#646 the problem was caused due to an incorrect translation of the bitwise-or operator.

For CPA#1130 (Listing 5.1) CPA-checker tried to prove the equality `x == ~y` by checking if the value-interval of x overlaps with the complemented value-interval of y. However this does not correctly capture the semantics of bitwise-negation.

```c
extern void __VERIFIER_error(void);
extern unsigned int __VERIFIER_nondet_int(void);

int main(){
  int x = __VERIFIER_nondet_int();
  int y = __VERIFIER_nondet_int();
  if(x == ~y){
            __VERIFIER_error();
  }
  return 0;
}
```

Listing 5.1: Test case triggering bug 1130 in CPA

```c
extern void __VERIFIER_error(void);
extern unsigned int __VERIFIER_nondet_uint(void);

int main(){
    unsigned int n = __VERIFIER_nondet_uint();
    if((0 % n) <= 100){
        __VERIFIER_error();
    }
    return 0;
}
```

Listing 5.2: Test case triggering bug 1194 in CPA

Meanwhile in CPA#1194 (Listing 5.2) the modelling of `%` did not cover the case where the left operand is zero, which caused it to miss the error.

5.3.2 Mishandling type-casts and overflows

Using the fact that unsigned casts are well-defined in C, Minotaur can generate test-cases where the only well-defined error trace must include an overflow. This can cause analyzers to miss the error, if overflows are not considered (see e.g. CPA#1132). More interestingly, we also found some cases where an analyzer would miss an error even though overflows are supported.

In Ultimate#642 (Listing 5.3) Ultimate would fail to compute an overflow for `(unsigned int) (uc + (4294967295*1))`, because the second operand of the shift following afterwards is constant.

In CPA#1114 the invariant generator would translate the potentially overflowing equality `if (3153284770U == x * 65599U)` into `x == 3153284770U / 65599U`. However these two equalities are not equivalent as the division might truncate and the multiplication might overflow.

```c
extern void __VERIFIER_error(void);
extern unsigned char __VERIFIER_nondet_uchar(void);
int main(){
  unsigned char uc = __VERIFIER_nondet_uchar();
  if((unsigned int)(((unsigned int)((unsigned int) uc) + (unsigned
   int)(4294967295*1)) >> ((unsigned int) 2)) < (unsigned int)  8){
    __VERIFIER_error();
  }
    return 0;
}
```

Listing 5.3: Test case triggering bug 642 in Ultimate

```c
extern void __VERIFIER_error(void);
extern unsigned short __VERIFIER_nondet_ushort(void);
int main(){
  unsigned short T2_460 = __VERIFIER_nondet_ushort();
  if((unsigned int) ((unsigned int) (((unsigned short)(T2_460)) + ((
   unsigned int) 0))) <= (unsigned int) ((unsigned int) 65536)){
    if(!((unsigned int) (((unsigned int) 112) - ((unsigned int) (((
   unsigned short)(T2_460)) + ((unsigned int) 72)))) < (unsigned int)
     ((unsigned int) 63))){
      __VERIFIER_error();
    }
  }
      return 0;
}
```

Listing 5.4: Test case bug 1363 in ESBMC

```c
extern signed char _VERIFIER_nondet_uchar(void);
extern void _MOPSA_assert(int);
int main(){
    signed char a = _MOPSA_rand_s8();
    if (242 + (unsigned char) a < 256) {
        _MOPSA_assert(242 + (unsigned char) a < 256);
    }
    return 0;
}
```

Listing 5.5: Test case triggering issue 150 in MOPSA

In ESBMC#1363 (Listing 5.4), when translating the expression `(unsigned int)T_2460 + 0 <= (unsigned int)65536`, the number `65536` would be cast to the type of `T_2460`, rather than the actual type of the expression it appears in. As the type of `T_2460` is `unsigned short`, this causes the number 65536 to overflow to 0. Therefore the inequality is mistranslated as $T\_2460 \leq 0$, which makes the second if guard unsatisfiable.

Type casts are also responsible for a curious precision issue in MOPSA (Listing 5.5, MOPSA#150), where due to over approximation of overflowing variables the assertion `242 + (unsigned char) a < 256` fails, despite being located in an if statement with the same condition.

5.4 Threats to validity

External: All mentioned bugs were reported and, with the exception of Symbiotic, were confirmed by the developers. We did not list cases where an error was missed due to an incorrect configuration (such as an incorrect unroll bound) of the tool[1] or undefined behaviour.

Internal: The current implementation of the translations could be partially incorrect. During early testing we found many unsound translations, which caused Minotaur to accidentally generate safe programs. However, these are easy to detect when testing multiple solvers, and haven't occurred on the newest version at the time of writing. Cases where the translation raises an error are covered by ensuring that at least a trivially unsafe program is generated.

Testing started with early versions of Minotaur, so tests generated with the current version might differ slightly from the reported ones. However we have managed to recreate all bugs (except ESBMC#1392) on the newest version of Minotaur.

[1]While some of the reported bugs depend on specific configuration options, the respective options are not supposed to have any impact on the soundness of the tool.

CHAPTER 6

Related work

Testing program analyzers: There have been several approaches that perform both differential [38] and metamorphic testing [72, 73] of program analyses. MCFuzz [72] and αDiff [38] have already been discussed in the introduction. Statifier [73] tests common static analyzers for Java programs and uses generated reports (rather than just TRUE/FALSE results) to guide transformations.

Techniques have also been developed for testing some of the components used by program analyzers. Not only are these worth testing in their own right, but they must also be correct for program analysis to remain correct. These include SMT solvers [45, 10, 67, 70, 71, 66], symbolic execution engines [36], abstract domains [11] and data flow analyses [61].

Compiler testing: Generating (well-defined) C programs has been an important part of compiler testing for many years. A common approach is to perform differential testing on randomly generated programs [43, 49, 50, 69]. Here it is especially important that programs are well-defined, as otherwise differences might just be down to unspecified behaviour.

Metamorphic testing [41, 42, 60, 20], often also referred to as equivalence modulo input (EMI) in the area, has also enjoyed great success, especially for finding silent miscompilations.

Zhang et. al. [74] present a third approach, which enumerates all swaps exchanges of variables in a given program and has also had some success at finding crashes and miscompilations.

Metamorphic testing: Metamorphic testing has further been applied to other domains, such as Datalog engines [44, 46], database systems [54, 56] and supervised classifiers [68]. Segura et.al. have collected a survey [57] on the topic.

CHAPTER 7

Conclusion

We have presented an approach to generate unsafe programs, which are populated with complex program expressions using SMT formulas. We use our knowledge about the satisfiability of these SMT formulas to guarantee that an error is reachable in the resulting programs. Our approach was able to find 10 soundness bugs in 5 state-of-the art program analyzers. Using a minimization technique we were able to report the bugs with small working examples, which led to them being fixed quickly by the developers.

Using the current parameters Minotaur seems to have reached a saturation point (as discussed by Livinksi et. al. [43]), with only two bugs having been found in the last few months. It is unclear whether this is due to the limitations of the approach itself, or whether changes to the experimental setup would be able to reveal further bugs.

7.1 Future work

There are many ways in which Minotaur can still be improved:

One option is to expand the number of generated language features, e.g. by implementing the SMT-LIB theory of floating points (which is unfortunately not currently supported by pySMT) and combinations of floating points, integers and bitvectors.

Testing itself could be also be improved. Timeouts could be adjusted dynamically to avoid long idle times on easy problems and high numbers of timeouts on hard ones.[1] A different choice of SMT seeds might help, as few SMT-COMP benchmarks are handcrafted and therefore most contain very similar files (thus leading to similarities in generated programs). Improving the parameter fuzzing for analyzers could trigger bugs in components which are not being tested at the moment.

[1]The difficulty of the seems to depend mainly on the SMT seed being used, making it hard to tune difficulty via other parameters.

Minotaur can already be used to generate safe programs (useful for precision tests), if an unsatisfiable SMT file is used as a seed. However both the seed mutation provided by STORM and the minimization procedure would need to be adapted accordingly, before performing extensive precision tests.

Minotaur is built on top of a benchmarking tool, so it should be possible to use it to generate benchmarks to measure analyzer performance. However further experiments would be necessary to see if there are any advantages over existing benchmarks.

Lastly it would be interesting to try and find ways to safely inject generated guards into broader classes of programs, which might find errors connected to program logic that is more complex than the one present in the programs currently generated by Minotaur.

List of Tables

List of Algorithms

Listings

Acronyms

AI	abstract interpretation. 4, 15
BMC	bounded model checking. 4, 15, 16
CEGAR	counterexample-guided abstraction refinement. 15
SAT	satisfiability problem over propositional formulas. 4
SMC	software model checker. 15
SMT	satisfiability modulo theories. 2, 4–7, 9, 10, 13, 15, 16, 25, 27, 28
SMT-COMP	satisfiability modulo theories comptetition. 5, 16, 27
SMT-LIB	satisfiability modulo theories library. 5, 10–12, 27
SV-COMP	competition on software verification. 7, 9, 16

Bibliography

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. „Automatic Predicate Abstraction of C Programs". In: *SIGPLAN Not.* 36.5 (2001), pp. 203–213. DOI: 10.1145/381694.378846.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. „The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525. DOI: 10.1109/TSE.2014.2372785.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *SMT-LIB - Logics*. URL: https://smtlib.cs.uiowa.edu/logics.shtml (visited on 12/31/2023).

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[5] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. „CVC4". In: *CAV'11*. Vol. 6806. LNCS. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[6] Dirk Beyer. *Reproducing SV-COMP Results (SV-COMP 2023)*. URL: https://sv-comp.sosy-lab.org/2023/systems.php (visited on 01/02/2024).

[7] Dirk Beyer. *Rules of the 12th Competition on Software Verification (SV-COMP 2023)*. URL: https://sv-comp.sosy-lab.org/2023/rules.php (visited on 01/05/2024).

[8] Dirk Beyer and M. Erkan Keremoglu. „CPAchecker: A Tool for Configurable Software Verification". In: *CAV'11*. Vol. 6806. LNCS. 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. „Symbolic Model Checking without BDDs". In: *TACAS'99*. Vol. 1579. LNCS. Springer Berlin Heidelberg, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[10] Alexandra Bugariu and Peter Müller. „Automatically testing string solvers". In: *ICSE '20*. ACM, 2020, pp. 1459–1470. DOI: 10.1145/3377811.3380398.

[11] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. „Automatically testing implementations of numerical abstract domains". In: *ASE'18'*. ACM, 2018, pp. 768–778. DOI: 10.1145/3238147.3240464.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. „KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs“. In: *OSDI'08*. USENIX Association, 2008, pp. 209–224. DOI: 10.5555/1855741.1855756.

[13] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. „SMTInterpol: An Interpolating SMT Solver“. In: *SPIN'12*. Vol. 7385. LNCS. Springer, 2012, pp. 248–254. DOI: 10.1007/978-3-642-31759-0_19.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. „The MathSAT5 SMT Solver“. In: *TACAS'13*. Vol. 7795. LNCS. Springer, 2013. DOI: 10.1007/978-3-642-36742-7_7.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. „Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications“. In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. „Counterexample-Guided Abstraction Refinement“. In: *CAV'00*. Vol. 1855. LNCS. Springer Berlin Heidelberg, 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[17] Edmund Clarke, Daniel Kroening, and Flavio Lerda. „A Tool for Checking ANSI-C Programs“. In: *TACAS'04*. Vol. 2988. LNCS. Springer Berlin Heidelberg, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[18] Edmund M. Clarke. „Model checking“. In: *FSTTCS'97*. Vol. 1346. LNCS. Springer Berlin Heidelberg, 1997, pp. 54–56. DOI: 10.1007/BFb0058022.

[19] Patrick Cousot and Radhia Cousot. „Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints“. In: *POPL'77*. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[20] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. „Automated Testing of Graphics Shader Compilers“. In: *Proc. ACM Program. Lang.* 1.OOPSLA'17 (2017). DOI: 10.1145/3133917.

[21] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. „Software Verification Using k-Induction“. In: *Static Analysis*. Vol. 6887. LNCS. Springer Berlin Heidelberg, 2011, pp. 351–368. DOI: 10.1007/978-3-642-23702-7_26.

[22] Kamil Dudka, Petr Peringer, and Tomás Vojnar. „Byte-Precise Verification of Low-Level List Manipulation“. In: *SAS'13*. Vol. 7935. LNCS. Springer, 2013, pp. 215–237. DOI: 10.1007/978-3-642-38856-9_13.

[23] Bruno Dutertre. „Yices 2.2“. In: *CAV'14*. Vol. 8559. LNCS. Springer, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[24] Evren Ermis, Alexander Nutz, Daniel Dietsch, Jochen Hoenicke, and Andreas Podelski. „Ultimate Kojak - (Competition Contribution)“. In: *TACAS'14*. LNCS. Springer, 2014, pp. 421–423. DOI: 10.1007/978-3-642-54862-8_36.

[25] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. „Sound sequentialization for concurrent program verification“. In: *PLDI'22*. ACM, 2022, pp. 506–521. DOI: 10.1145/3519939.3523727.

[26] Jochen Hoenicke François Bobot Martin Bromberger. *18th International Satisfiability Modulo Theories Competition (SMT-COMP 2023)*. https://clc-gitlab.cs.uiowa.edu:2443/explore/projects. 2023.

[27] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. „ESBMC 5.0: An Industrial-Strength C Model Checker“. In: *ASE'18*. ACM, 2018. DOI: 10.1145/3238147.3240481.

[28] Marco Gario and Andrea Micheli. „PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms“. In: *SMT Workshop'15*. 2015.

[29] Susanne Graf and Hassen Saidi. „Construction of abstract state graphs with PVS“. In: *CAV'97*. Vol. 1254. LNCS. Springer Berlin Heidelberg, 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10.

[30] Marius Greitschus, Daniel Dietsch, and Andreas Podelski. „Loop Invariants from Counterexamples“. In: *SAS'17*. Vol. 10422. LNCS. Springer, 2017, pp. 128–147. DOI: 10.1007/978-3-319-66706-5_7.

[31] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. „Swarm Testing“. In: *ISSTA'12*. ACM, 2012, pp. 78–88. DOI: 10.1145/2338965.2336763.

[32] Arie Gurfinkel, Temesghen Kahsai, and Jorge A Navas. „SeaHorn: A framework for verifying C programs (competition contribution)“. In: *TACAS'15*. Vol. 9035. LNCS. Springer Berline Heidelberg, 2015, pp. 447–450. DOI: https://doi.org/10.1007/978-3-662-46681-0_41.

[33] Sang Kil Cha Haeun Lee Soomin Kim. „Fuzzle: Making a Puzzle for Fuzzers“. In: ASE '22. IEEE/ACM, 2022. DOI: https://doi.org/10.1145/3551349.3556908.

[34] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. „Refinement of Trace Abstraction“. In: *SAS'09*. LNCS. Springer, 2009, pp. 69–85. DOI: 10.1007/978-3-642-03237-0_7.

[35] M. Journault and A. Miné. „Static analysis by abstract interpretation of the functional correctness of matrix manipulating programs“. In: *SAS'16*. Vol. 9837. LNCS. http://www-apr.lip6.fr/~mine/publi/article-journault-al-sas16.pdf. Springer, 2016, pp. 257–277. DOI: 10.1007/978-3-662-53413-7_13.

[36] Timotej Kapus and Cristian Cadar. „Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing“. In: *ASE '17*. IEEE Press, 2017, pp. 590–600. DOI: 10.1109/ASE.2017.8115669.

[37] James C. King. „Symbolic Execution and Program Testing“. In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[38] Christian Klinger, Maria Christakis, and Valentin Wüstholz. „Differentially Testing Soundness and Precision of Program Analyzers“. In: *ISSTA'19*. ACM, 2019, pp. 239–250. DOI: 10.1145/3293882.3330553.

[39] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. „SMT-Based Model Checking for Recursive Programs“. In: *CAV'14*. Vol. 8559. LNCS. Springer International Publishing, 2014, pp. 17–34. DOI: 10.1007/978-3-319-08867-9_2.

[40] William Landi. „Undecidability of Static Analysis“. In: *ACM Lett. Program. Lang. Syst.* 1.4 (1992), pp. 323–337. DOI: 10.1145/161494.161501.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. „Compiler Validation via Equivalence modulo Inputs“. In: *SIGPLAN Not.* 49.6 (2014), pp. 216–226. URL: https://doi.org/10.1145/2666356.2594334.

[42] Vu Le, Chengnian Sun, and Zhendong Su. „Finding Deep Compiler Bugs via Guided Stochastic Program Mutation“. In: *SIGPLAN Not.* 50.10 (2015), pp. 386–399. DOI: 10.1145/2858965.2814319.

[43] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. „Random Testing for C and C++ Compilers with YARPGen“. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428264.

[44] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. „Metamorphic Testing of Datalog Engines“. In: ESEC/FSE'21. ACM, 2021, pp. 639–650. DOI: 10.1145/3468264.3468573.

[45] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. „Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing“. In: ESEC/FSE'20. ACM, 2020, pp. 701–712. DOI: 10.1145/3368089.3409763.

[46] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. „Dependency-Aware Metamorphic Testing of Datalog Engines“. In: *ISSTA'23*. ACM, 2021, pp. 236–247. DOI: 10.1145/3597926.3598052.

[47] A. Miné, A. Ouadjaout, and M. Journault. „Design of a modular platform for static analysis“. In: *TAPAS'18*. LNCS. http://www-apr.lip6.fr/~mine/publi/mine-al-tapas18.pdf. 2018.

[48] Leonardo de Moura and Nikolaj Bjørner. „Z3: An Efficient SMT Solver“. In: *TACAS'08*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[49] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. „Random testing of C compilers targeting arithmetic optimization“. In: *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 2012, pp. 48–53.

[50] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. „Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions“. In: *IPSJ Trans. Syst. LSI Des. Methodol.* 7 (2014), pp. 91–100. DOI: 10.2197/IPSJTSLDM.7.91.

[51] *National Vulnerability Database*. URL: https://nvd.nist.gov/ (visited on 01/05/2024).

[52] Greg Nelson and Derek C. Oppen. „Simplification by Cooperating Decision Procedures“. In: *ACM Trans. Program. Lang. Syst.* 1.2 (1979), pp. 245–257. DOI: 10.1145/357073.357079.

[53] Aina Niemetz and Mathias Preiner. „Bitwuzla“. In: *CAV'23*. Vol. 13965. LNCS. Springer, 2023, pp. 3–17. DOI: 10.1007/978-3-031-37703-7_1.

[54] Manuel Rigger and Zhendong Su. „Finding Bugs in Database Systems via Query Partitioning“. In: *OOPSLA'20*. Vol. 4. Proc. ACM Program. Lang. ACM, 2020. DOI: 10.1145/3428279.

[55] Peter Schrammel and Daniel Kroening. „2LS for Program Analysis“. In: *TACAS'16*. Springer Berlin Heidelberg, 2016, pp. 905–907. DOI: 10.1007/978-3-662-49674-9_56.

[56] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. „Automated Generation of Metamorphic Relations for Query-Based Systems“. In: MET '22. ACM, 2022, pp. 48–55. DOI: 10.1145/3524846.3527338.

[57] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. „A Survey on Metamorphic Testing“. In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 805–824. DOI: 10.1109/TSE.2016.2532875.

[58] Jiří Slabý, Jan Strejček, and Marek Trtík. „Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution“. In: *FMICS'12*. Springer Berlin Heidelberg, 2012, pp. 207–221. DOI: 10.1007/978-3-642-32469-7_14.

[59] *SMT-COMP - Participants*. URL: https://smt-comp.github.io/2023/participants.html (visited on 12/31/2023).

[60] Chengnian Sun, Vu Le, and Zhendong Su. „Finding Compiler Bugs via Live Code Mutation“. In: *SIGPLAN Not.* 51.10 (2016), pp. 849–863. ISSN: 0362-1340. DOI: 10.1145/3022671.2984038.

[61] Jubi Taneja, Zhengyang Liu, and John Regehr. „Testing static analyses for precision and soundness“. In: *CGO'20'*. ACM, 2020, pp. 81–93. DOI: 10.1145/3368826.3377927.

[62] Cesare Tinelli. „A DPLL-Based Calculus for Ground Satisfiability Modulo Theories“. In: *Logics in Artificial Intelligence (JELIA'02)*. Vol. 2424. LNCS. Springer Berlin Heidelberg, 2002, pp. 308–319. DOI: 10.1007/3-540-45757-7_26.

[63] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. „The SMT Competition 2015-2018“. In: *J. Satisf. Boolean Model. Comput.* 11.1 (2019), pp. 221–259. DOI: 10.3233/SAT190123.

[64] Philipp Wendler. *CPA Issue* #1114. URL: https://gitlab.com/sosy-lab/software/cpachecker/-/issues/1114#note_1478681483 (visited on 12/14/2023).

[65] Philipp Wendler. *CPA Issue* #1194. URL: https://gitlab.com/sosy-lab/software/cpachecker/-/issues/1194#note_1688812945 (visited on 12/14/2023).

[66] Dominik Winterer, Chengyu Zhang, and Zhendong Su. „On the unusual effectiveness of type-aware operator mutations for testing SMT solvers“. In: *OOPSLA'20*. Vol. 4. Proceedings of the ACM on Programming Languages. 2020, pp. 1–25. DOI: 10.1145/3428261.

[67] Dominik Winterer, Chengyu Zhang, and Zhendong Su. „Validating SMT solvers via semantic fusion“. In: *PLDI'20*. ACM, 2020, pp. 718–730. DOI: 10.1145/3385412.3385985.

[68] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Chen. „Application of Metamorphic Testing to Supervised Classifiers“. In: *QSIC'09* (2009), pp. 135–144. DOI: 10.1109/QSIC.2009.26.

[69] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. „Finding and Understanding Bugs in C Compilers“. In: *PLDI '11*. Vol. 46. ACM SIGPLAN Notices. ACM, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.

[70] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. „Fuzzing SMT solvers via two-dimensional input space exploration“. In: *ISSTA '21*. ACM, 2021, pp. 322–335. DOI: 10.1145/3460319.3464803.

[71] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. „Skeletal Approximation Enumeration for SMT Solver Testing“. In: *ESEC/FSE'21*. ACM, 2021, pp. 1141–1153. DOI: 10.1145/3468264.3468540.

[72] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. „Finding and Understanding Bugs in Software Model Checkers“. In: *ESEC/FSE'19*. ACM, 2019, pp. 763–773. DOI: 10.1145/3338906.3338932.

[73] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. „Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations“. In: ESEC/FSE'23. ACM, 2023, pp. 237–249. DOI: 10.1145/3611643.3616272.

[74] Qirun Zhang, Chengnian Sun, and Zhendong Su. „Skeletal program enumeration for rigorous compiler testing“. In: *PLDI'17*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 347–361. DOI: 10.1145/3062341.3062379.

[75] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. „Metamorphic testing and its applications“. In: *ISFST'04*. Software Engineers Association Xian, China. 2004, pp. 346–351.